Recursion

--by Harsh Gupta

Goal

- Understand recursion
- Understand and create recursive trees
- Understand applications of recursion
- Assess time complexity of recursive algorithms

Recap On Functions

- A function is a block of code which runs the code inside with the parameters it is given.

```
int add(int a, int b) {
    return a + b;
}
```

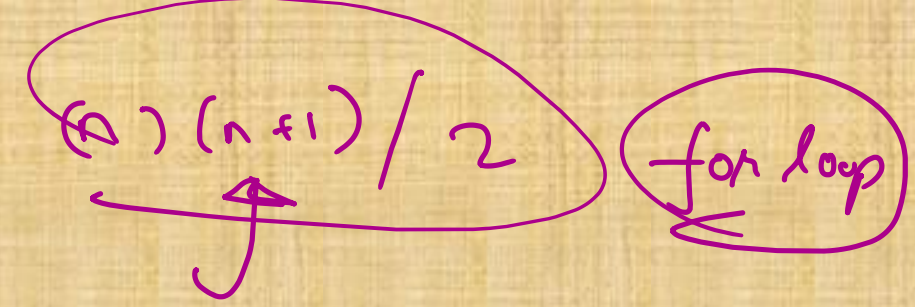
What is Recursion?

Recursion happens when a function calls itself on a different set of input parameters.

Used when the solution for current problem involves first solving a smaller sub-problem.

```
int f(int n){
    return n*2;
}
```

```
int f(int n) {
    int ans= f(n-2)
}
```

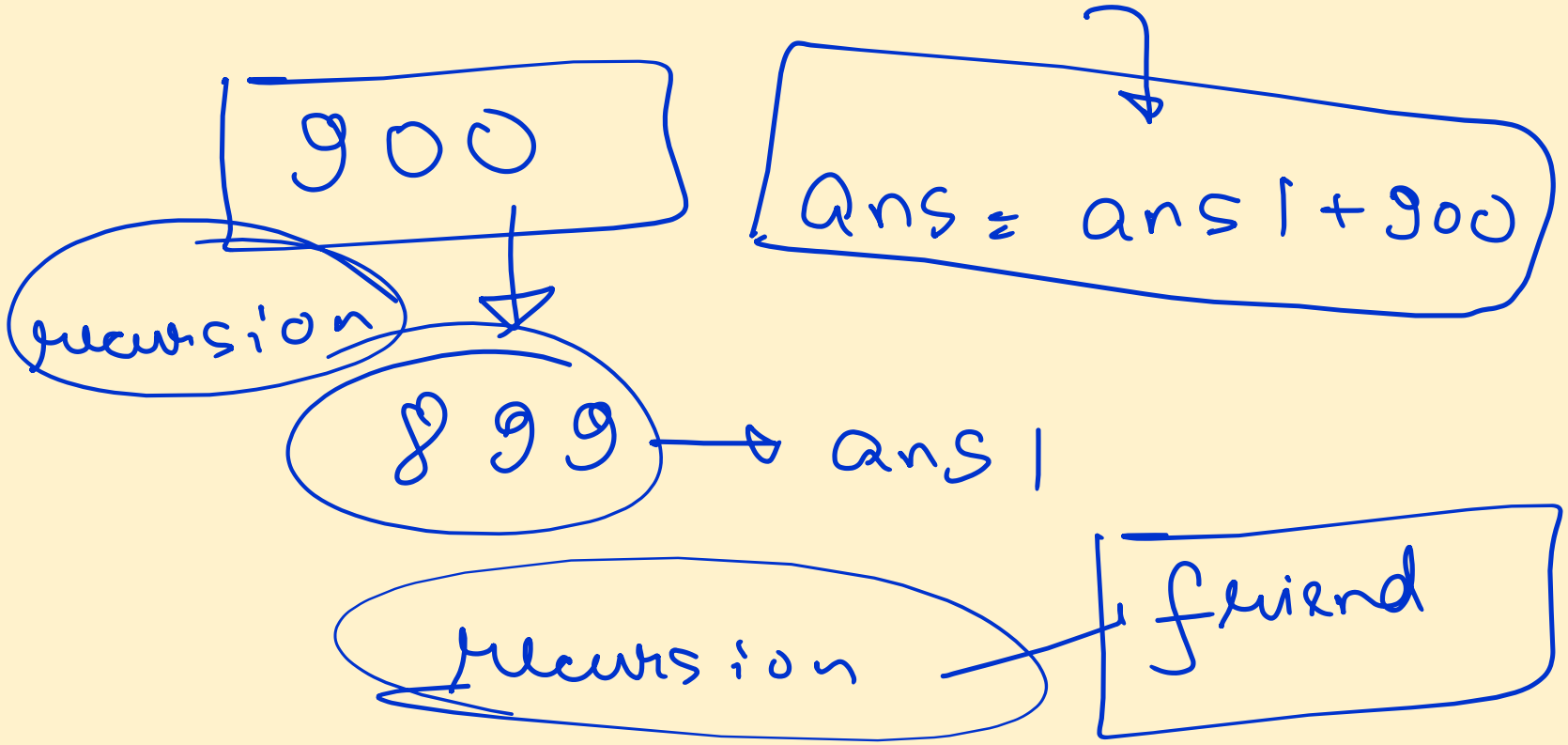



```
int sum_0_to_n(int n) {
    if (n <= 0) return 0;
    return sum_0_to_n(n-1) + n;
}
```

base case

The above function will find the sum from 0 to the given parameter.

Think that what minimal task you need to do.



899 + 900

892 + 899 + 900

1 + 2 + 3 + 4 + 5

10

900

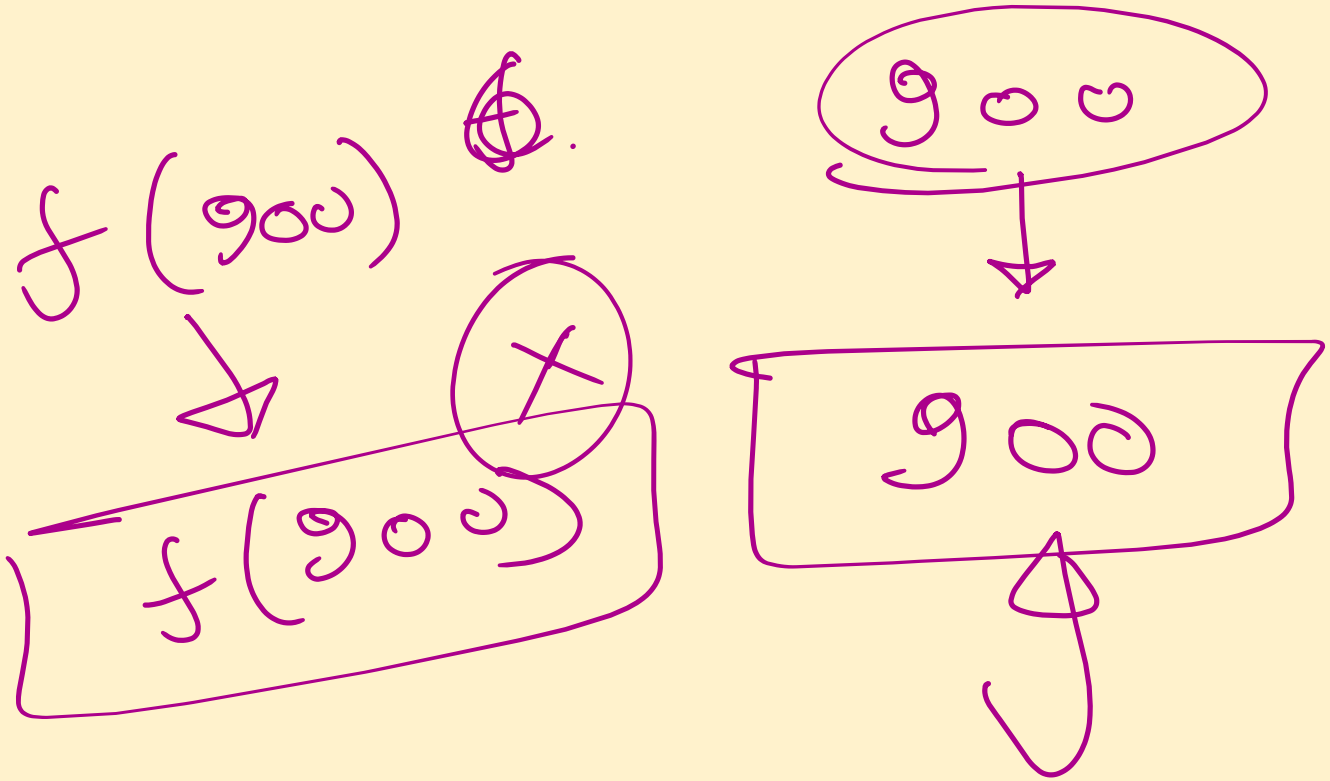
899

+900

10 + 5 = 15

900

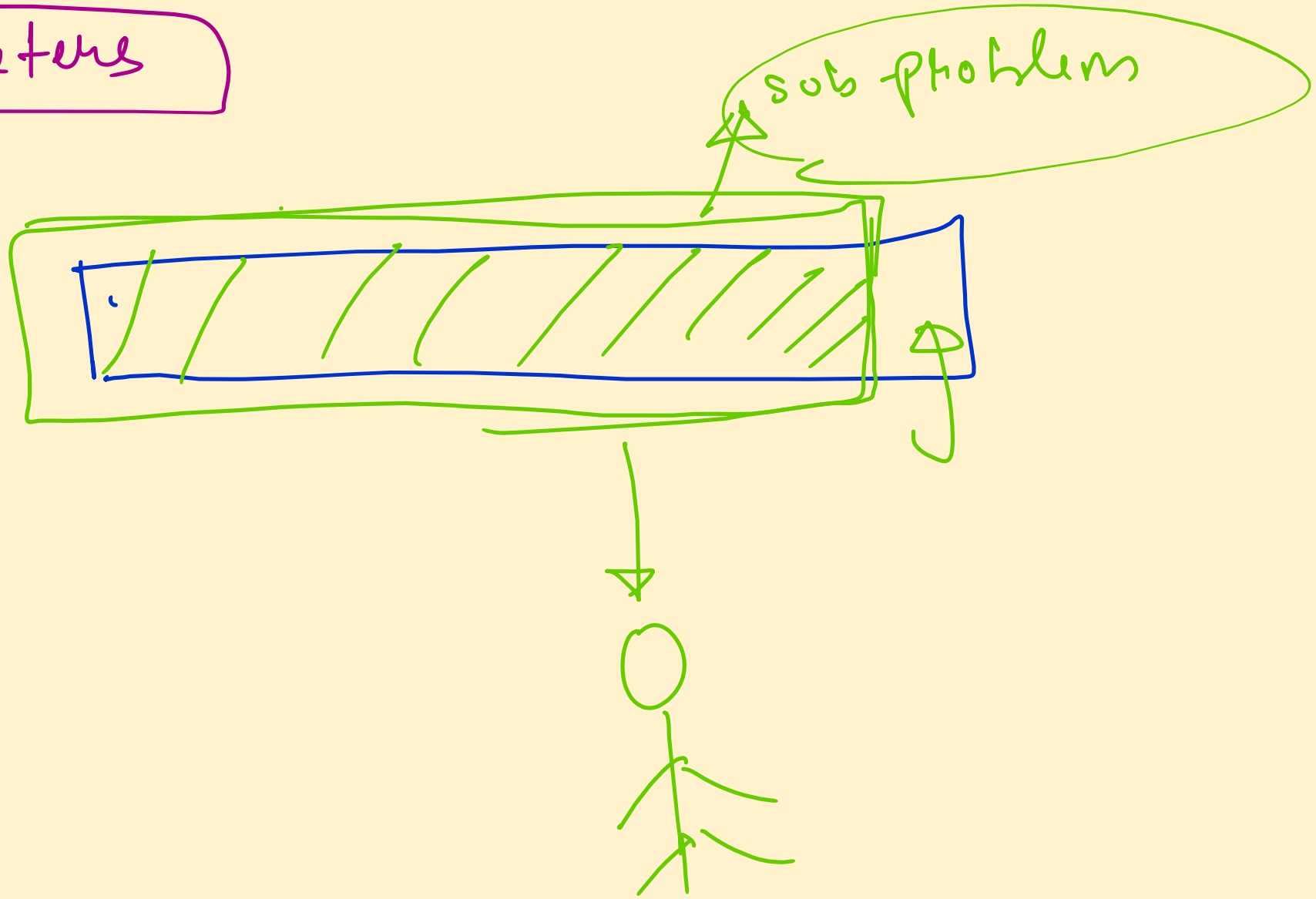
8


f(900)
f(900)
900
900

Basic Structure of a Recursive Function

- Parameters to start the function
- Appropriate base case(s) to end the recursion (Why ??)
- Recursively solve the sub-problems
- Process the result and return the value

Parameters



f (parameters) {

base cases

code

recursion calls

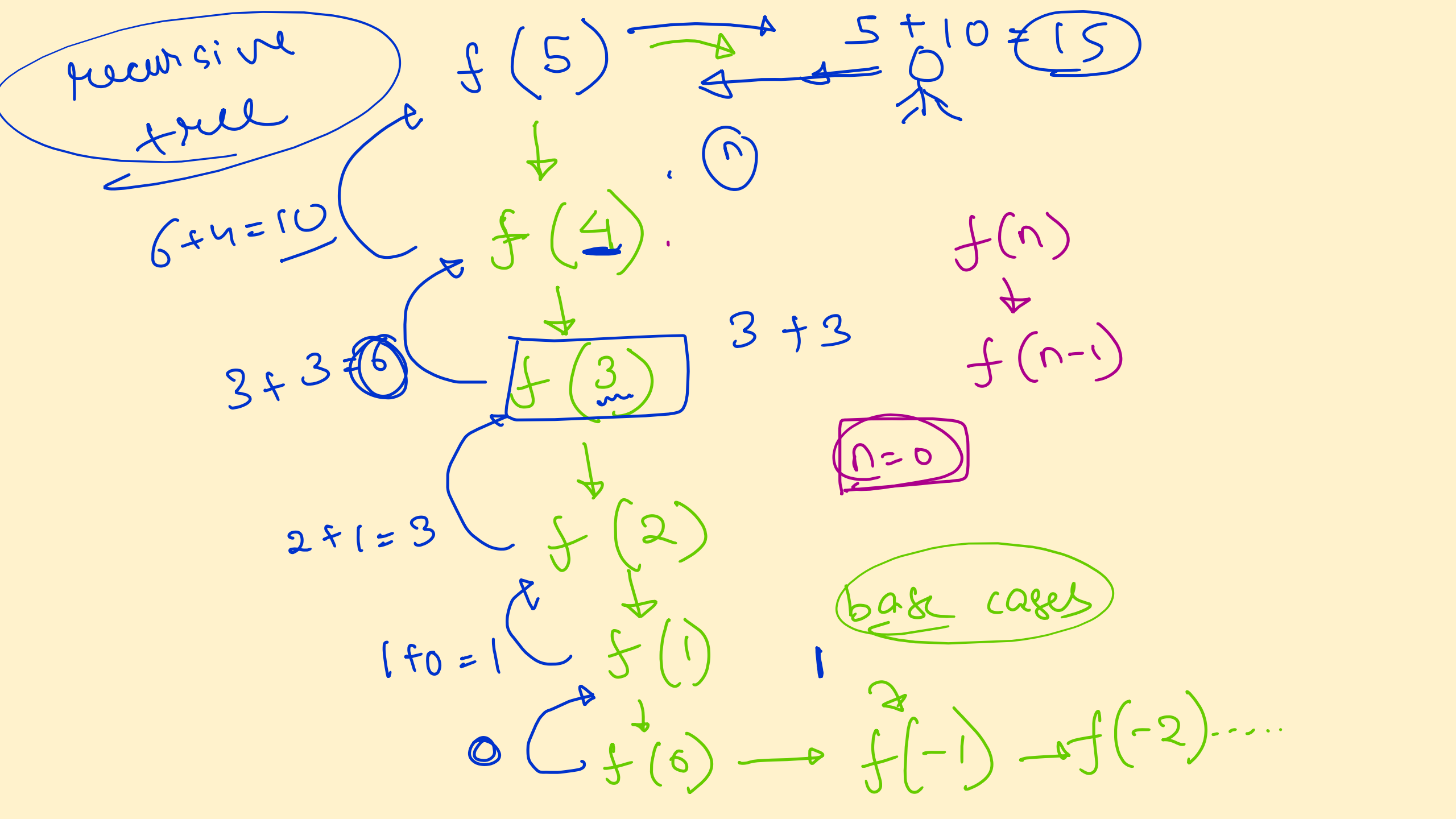
code

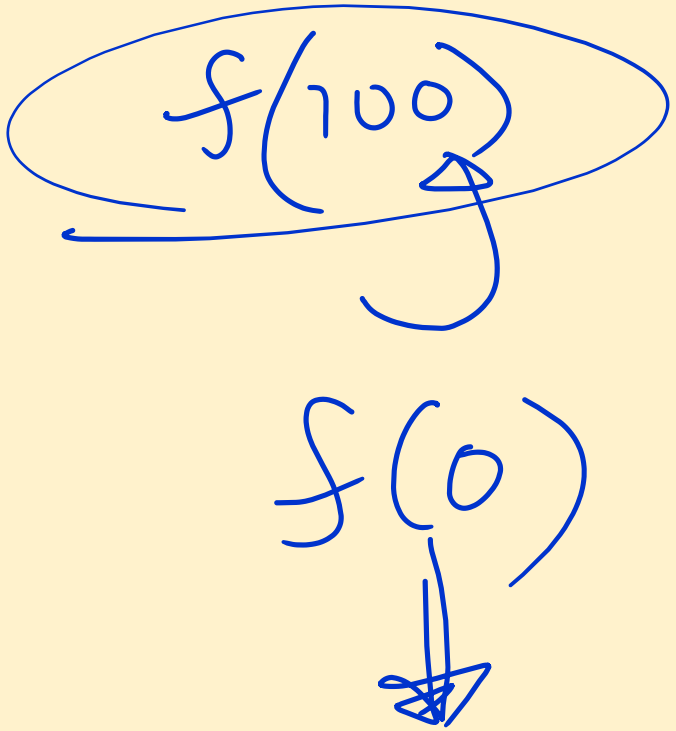
}

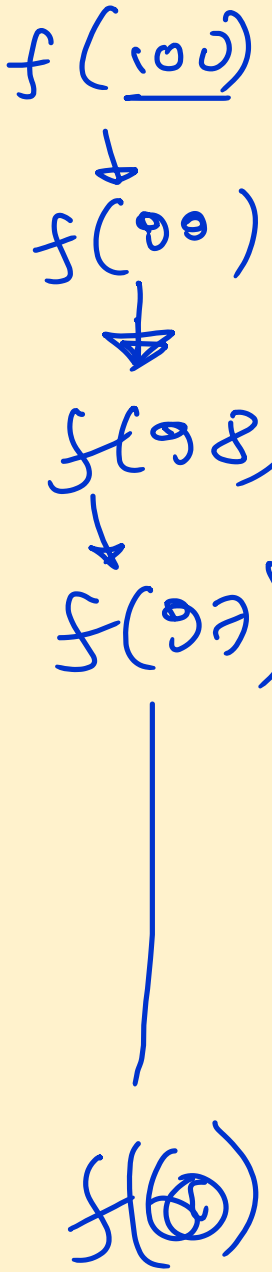
Recursive Tree

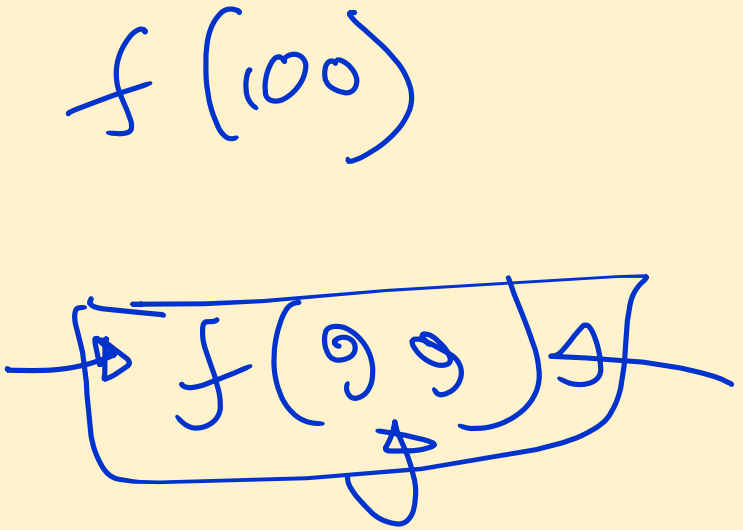
A recursive tree is similar to a “mind map” of the function call.

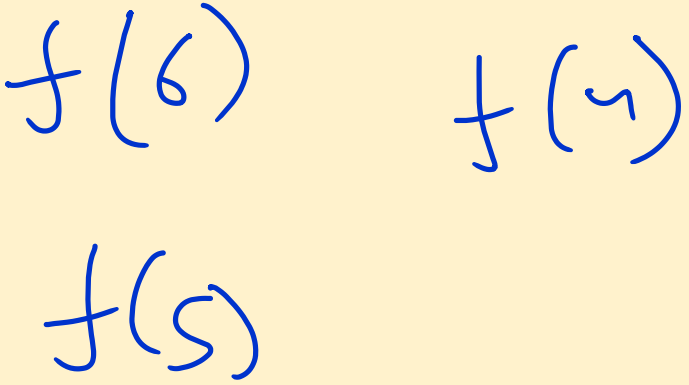
Recursive trees are useful to help us understand how the function acts.


recursive tree
f(5)
5 + 10 = 15
n
6 + 4 = 10
f(4)
f(n)
f(n-1)
3 + 3
3 + 3 = 6
f(3)
n = 0
2 + 1 = 3
f(2)
1 + 0 = 1
f(1)
base cases
1
0
f(0)
f(-1)
f(-2)


f(100)
f(0)


f(100)
f(99)
f(98)
f(97)
f(6)


f(100)
f(99)


f(6)
f(5)
f(4)

f(900)

```
int f ( int n ) {
    base case → if (n==900) return 900
    return f(n+1) + n;
}
```

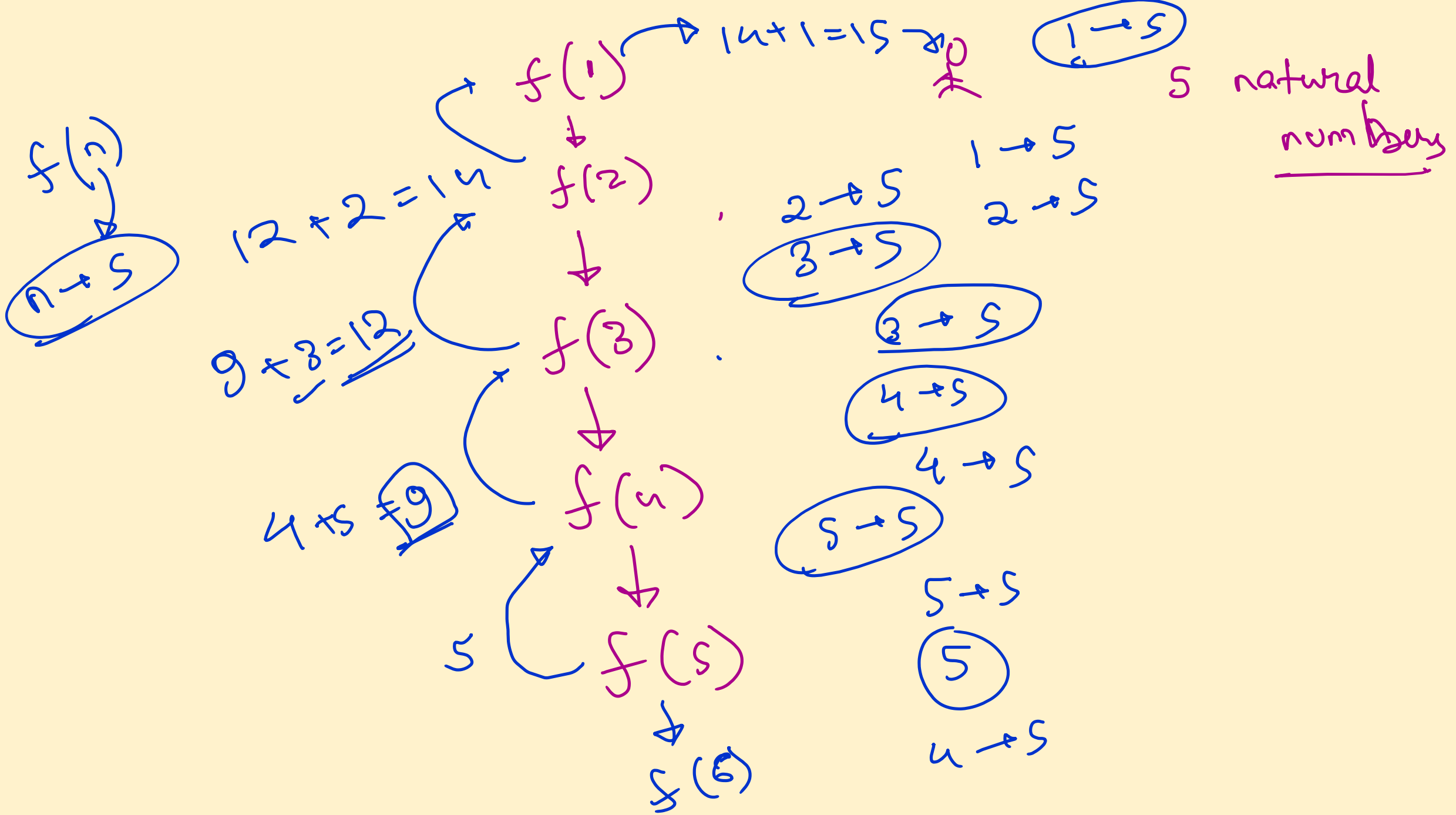
n 900

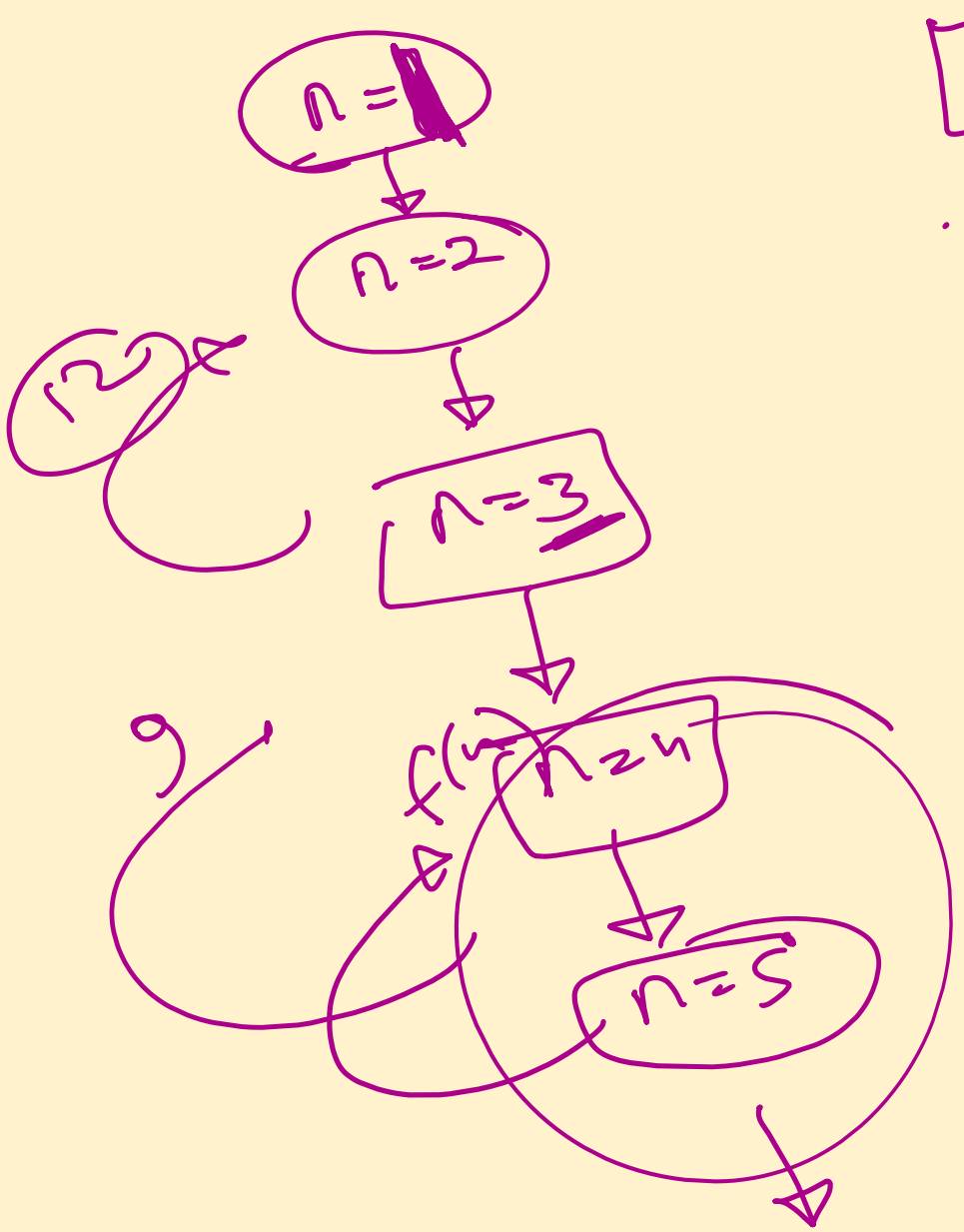
```
int f(int n 9. int a) {
    if(n==a) return n;
    return f(n+1) + n;
}
```

②

100

f(1, 100)


f(1)
14+1=15
1 → 5
5 natural numbers
f(2)
12+2=14
1 → 5
2 → 5
2 → 5
f(3)
9+3=12
3 → 5
3 → 5
f(4)
4+5=9
4 → 5
4 → 5
f(5)
5
5 → 5
5
f(6)
4 → 5
f(n)
n → 5



S

S

int f (int n) {

if (n == 1) return 1;

return f(n+1) + n;

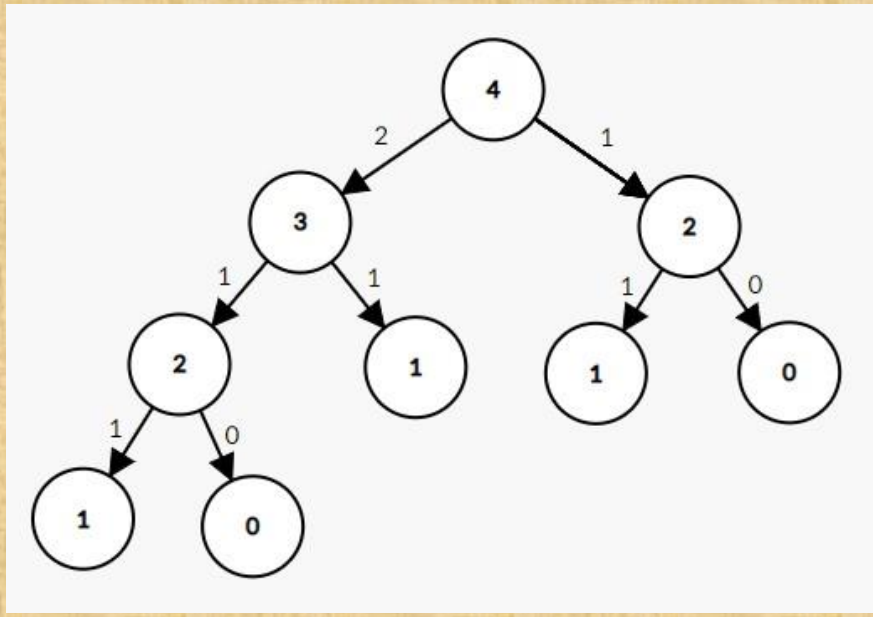
f(5) → f(4) → f(3)

Example

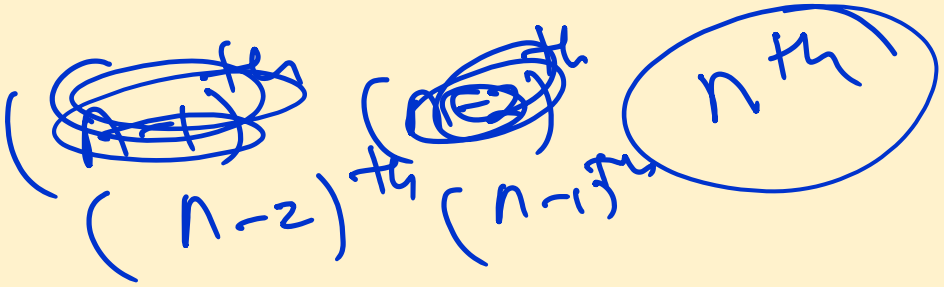
Fibonacci function:

```
int fib(int n) {
    if (n == 0) return 0;
    if (n == 1) return 1;
    return fib(n-1) + fib(n-2);
}
```

Recursive tree:

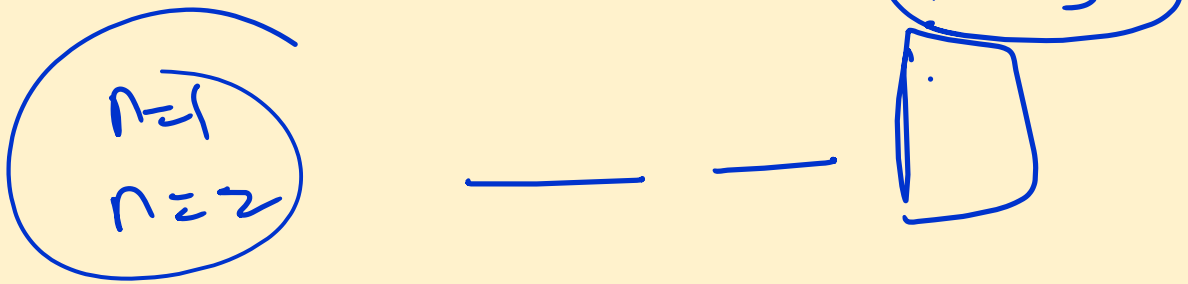


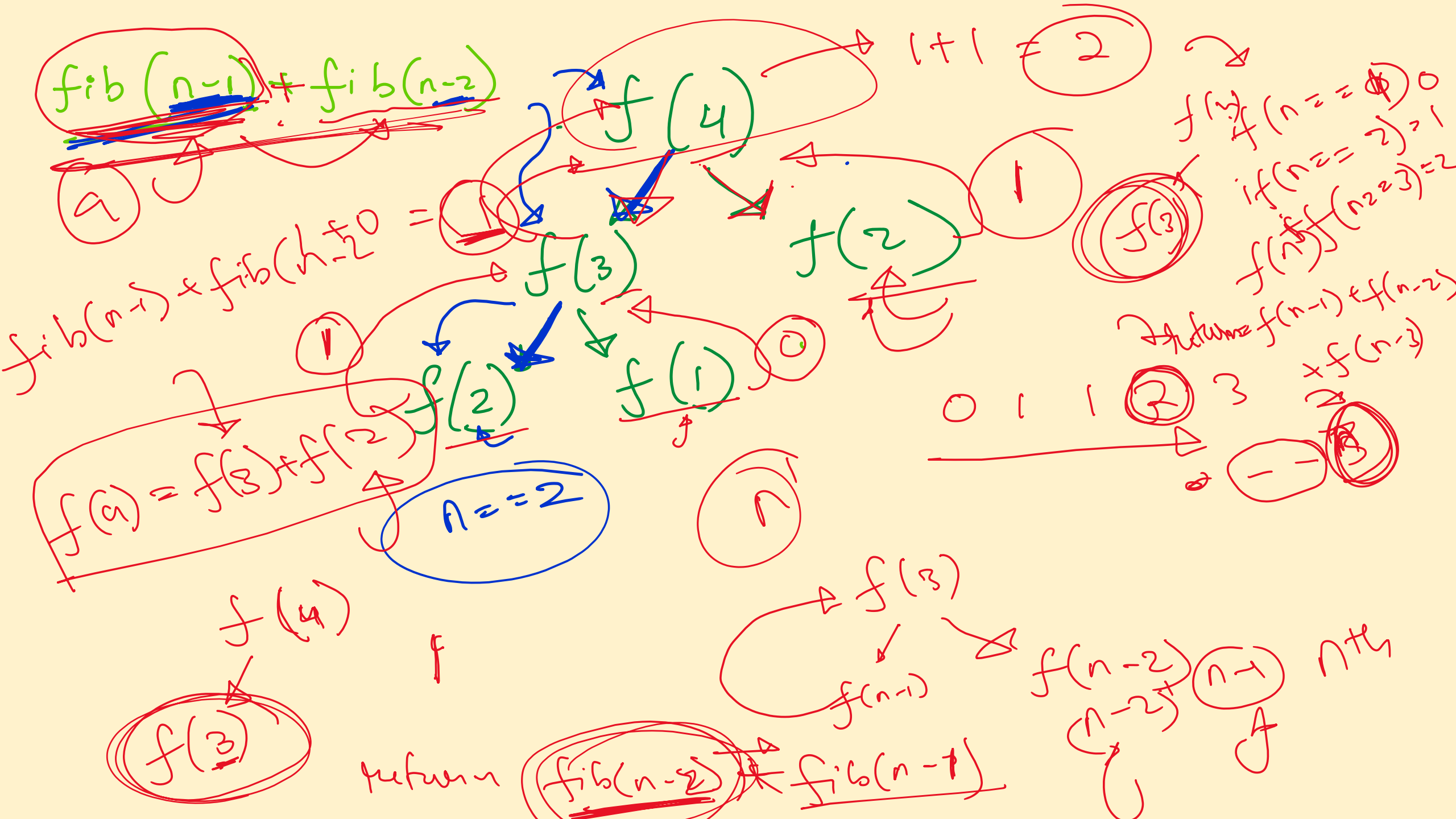
nth fibo using recursion

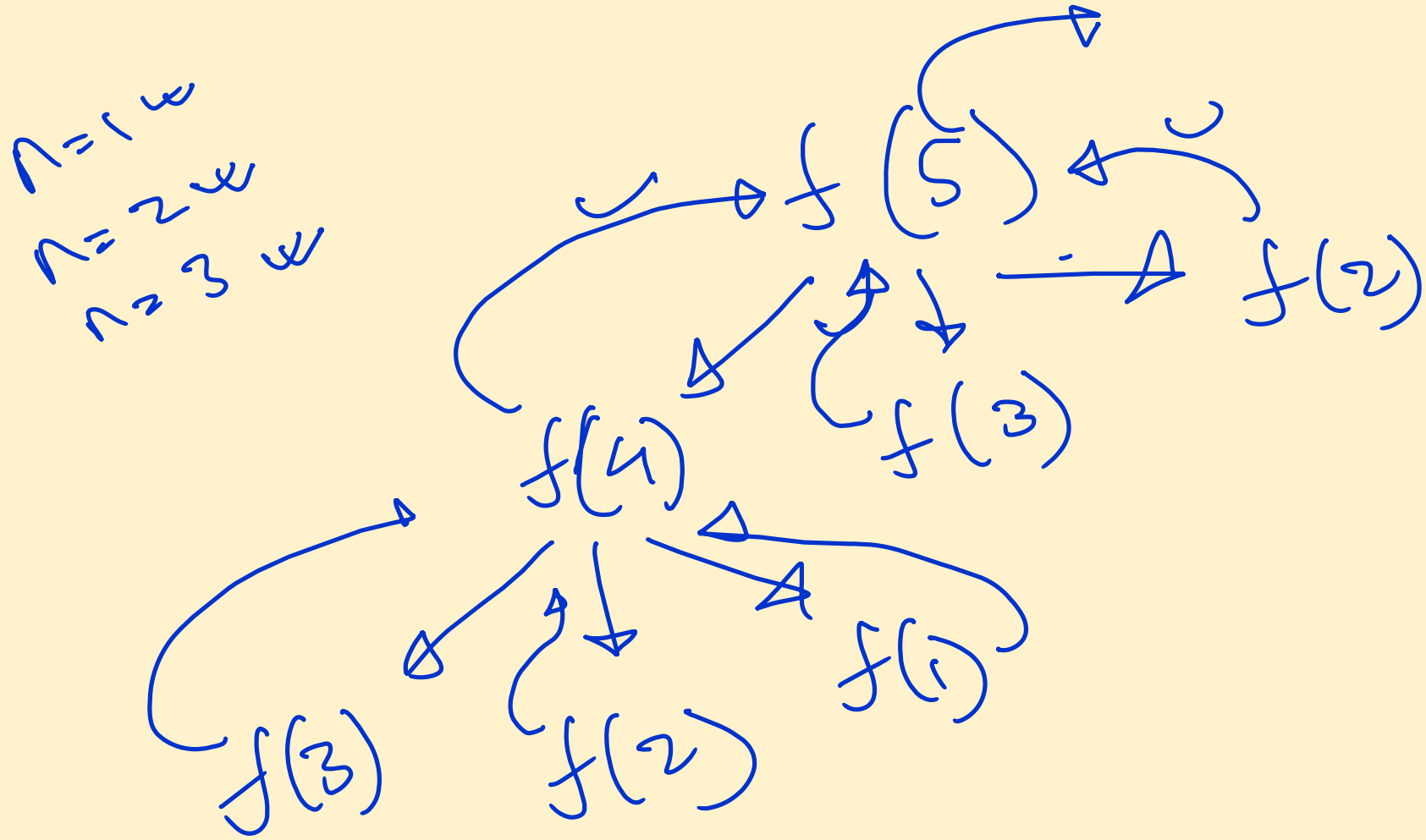


0 1

```
int fib(int n) {
    if(n==0) return 0;
    if(n==2) return 1;
    return fib(n-1) + fib(n-2);
}
```



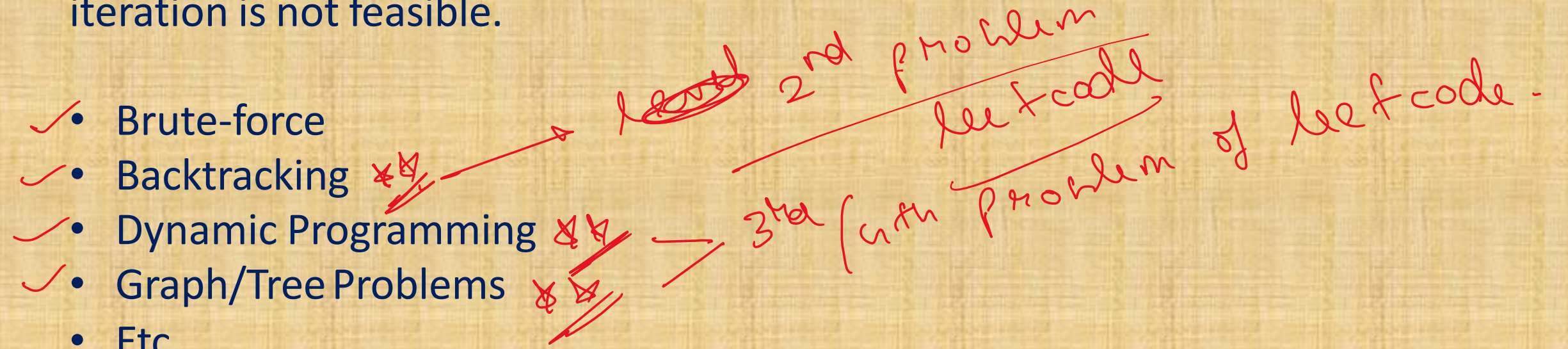

fib(n-1) + fib(n-2)
f(4)
f(3)
f(2)
f(1)
f(2)
f(3)
f(3)
f(n-2)
n-1
nth
fib(n-2)
fib(n-1)
return


f(5)
f(2)
f(3)
f(4)
f(3)
f(2)
f(1)

When do we need recursion?

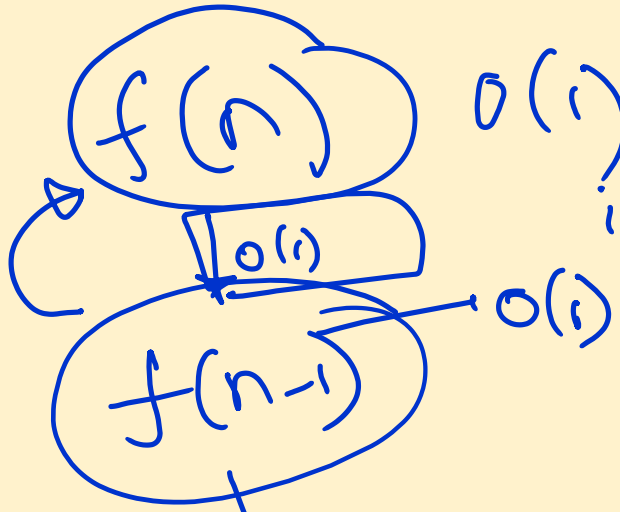
Recursion is usually used in complex situations where iteration is not feasible.

- Brute-force
- Backtracking
- Dynamic Programming
- Graph/Tree Problems
- Etc.



Time complexity of Recursion

$f(n) = f(n-1) + n$

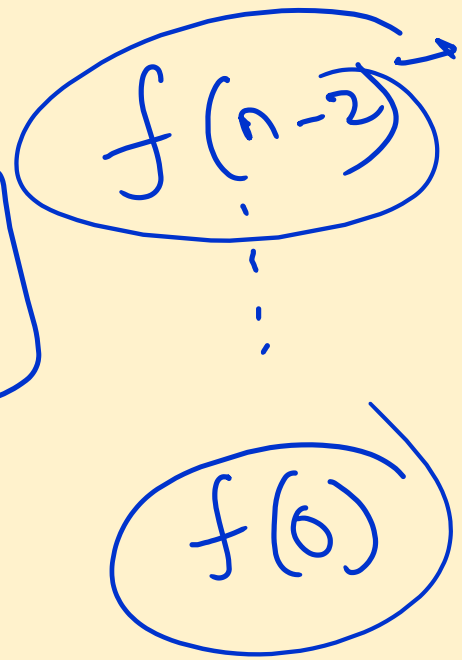



```
int f(int n) {
    ____________
    return f(n-1) + n;
```

O(n)

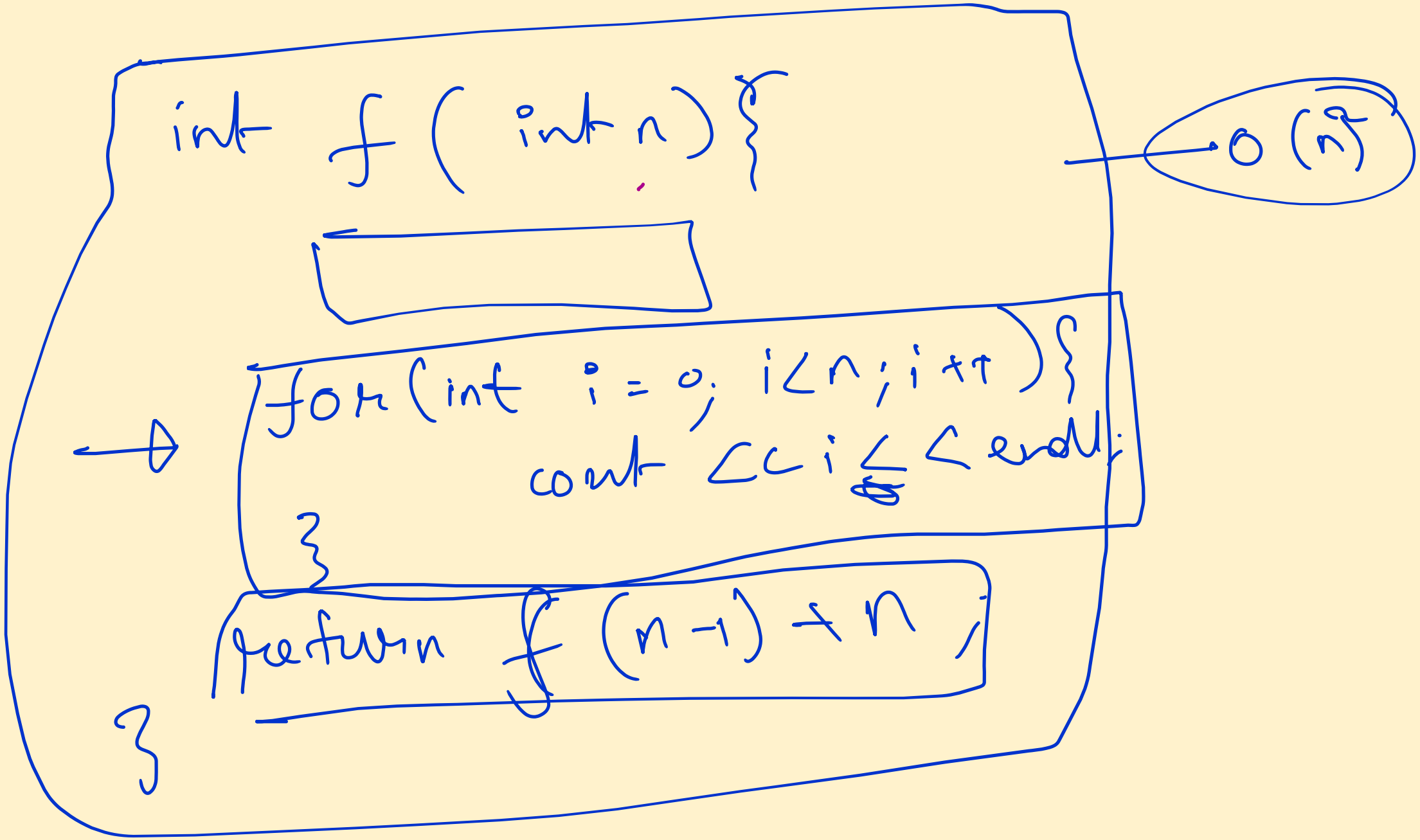
1

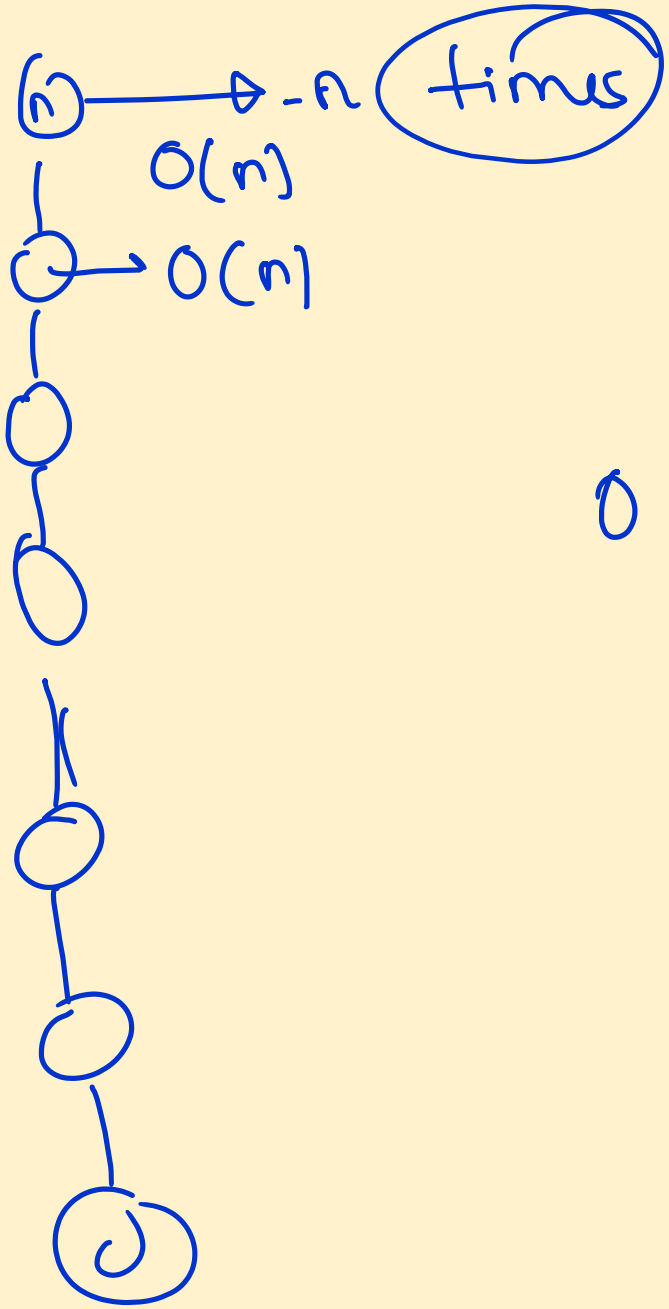
O(1) × n = O(n)



O(n) + O(n) + ... n times

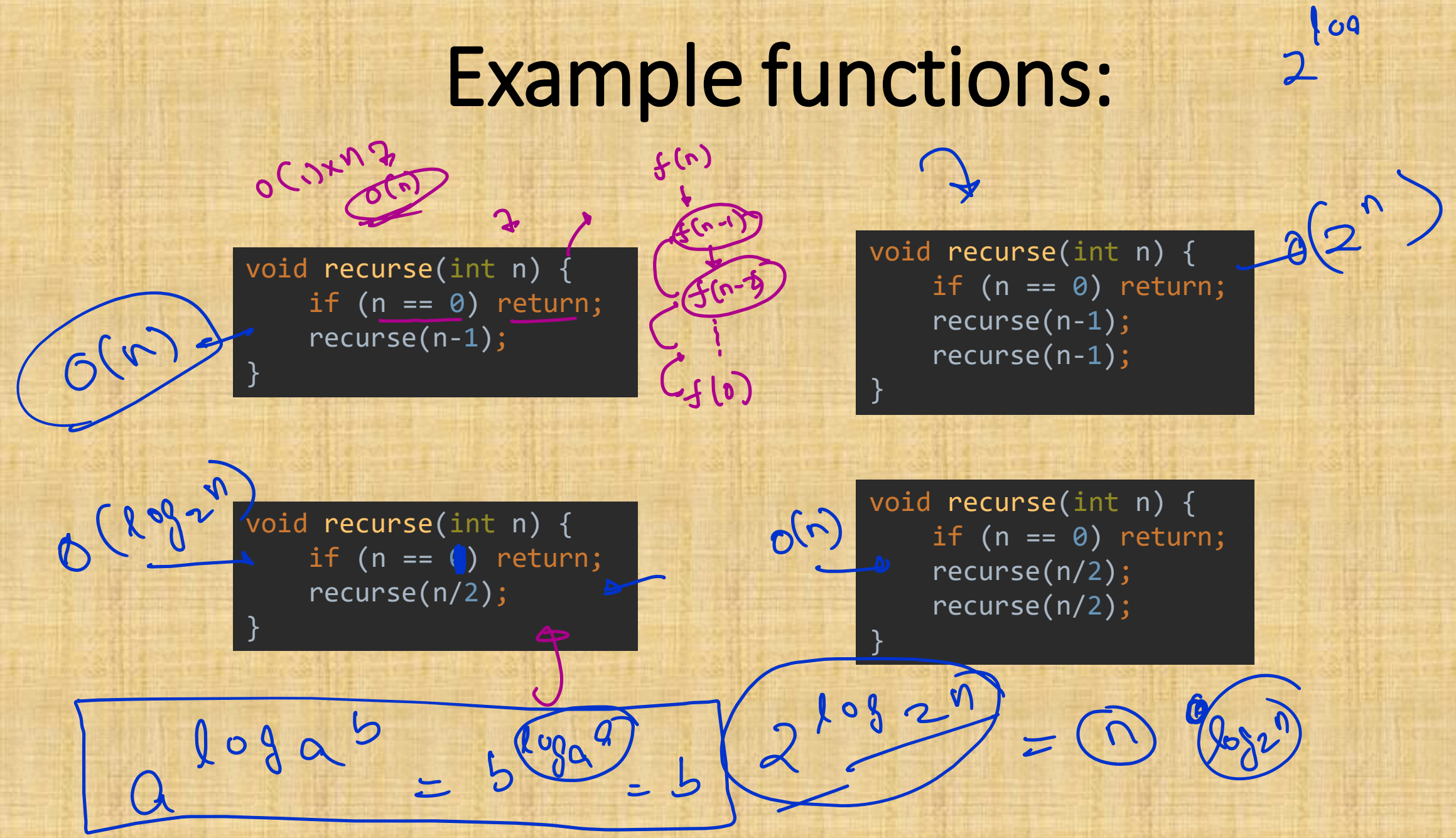
$O(n^2)$


int f (int n) {
O (n)
for(int i = 0; i<n; i++) {
cout << i << endl;
}
return f (n-1) + n ;
}



$O(n) \times n = O(n^2)$

Example functions:

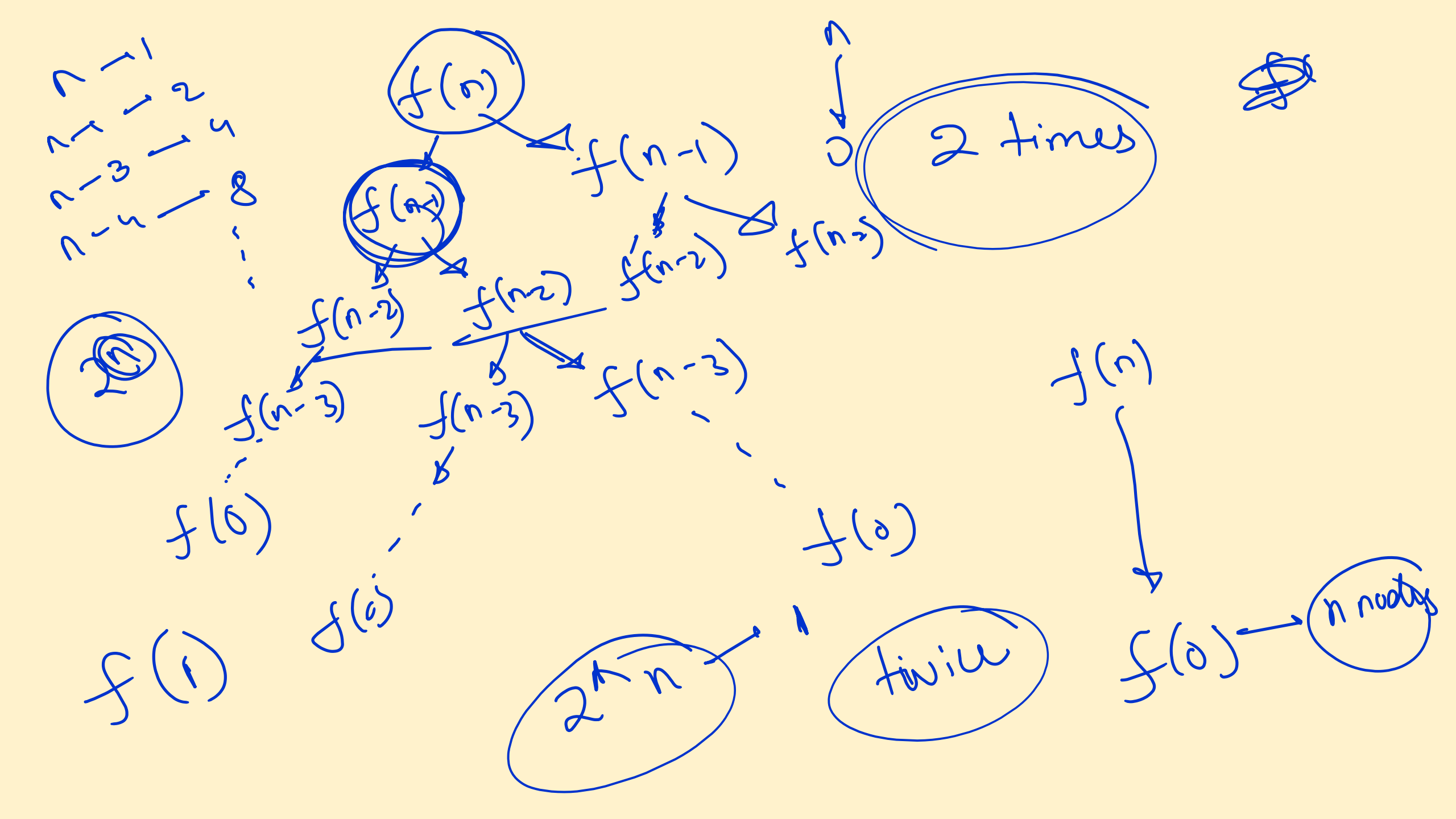


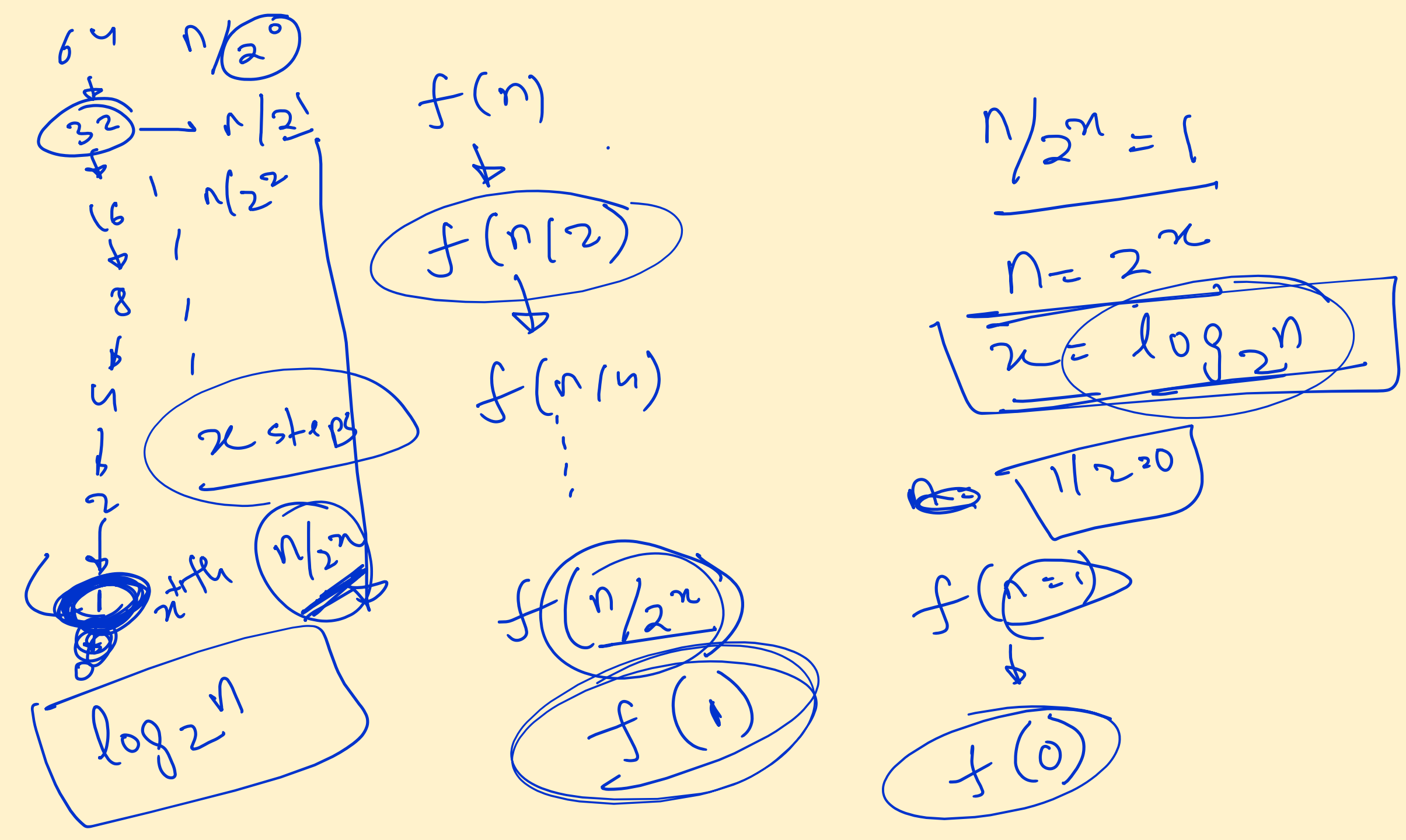
```
void recurse(int n) {
    if (n == 0) return;
    recurse(n-1);
}
```

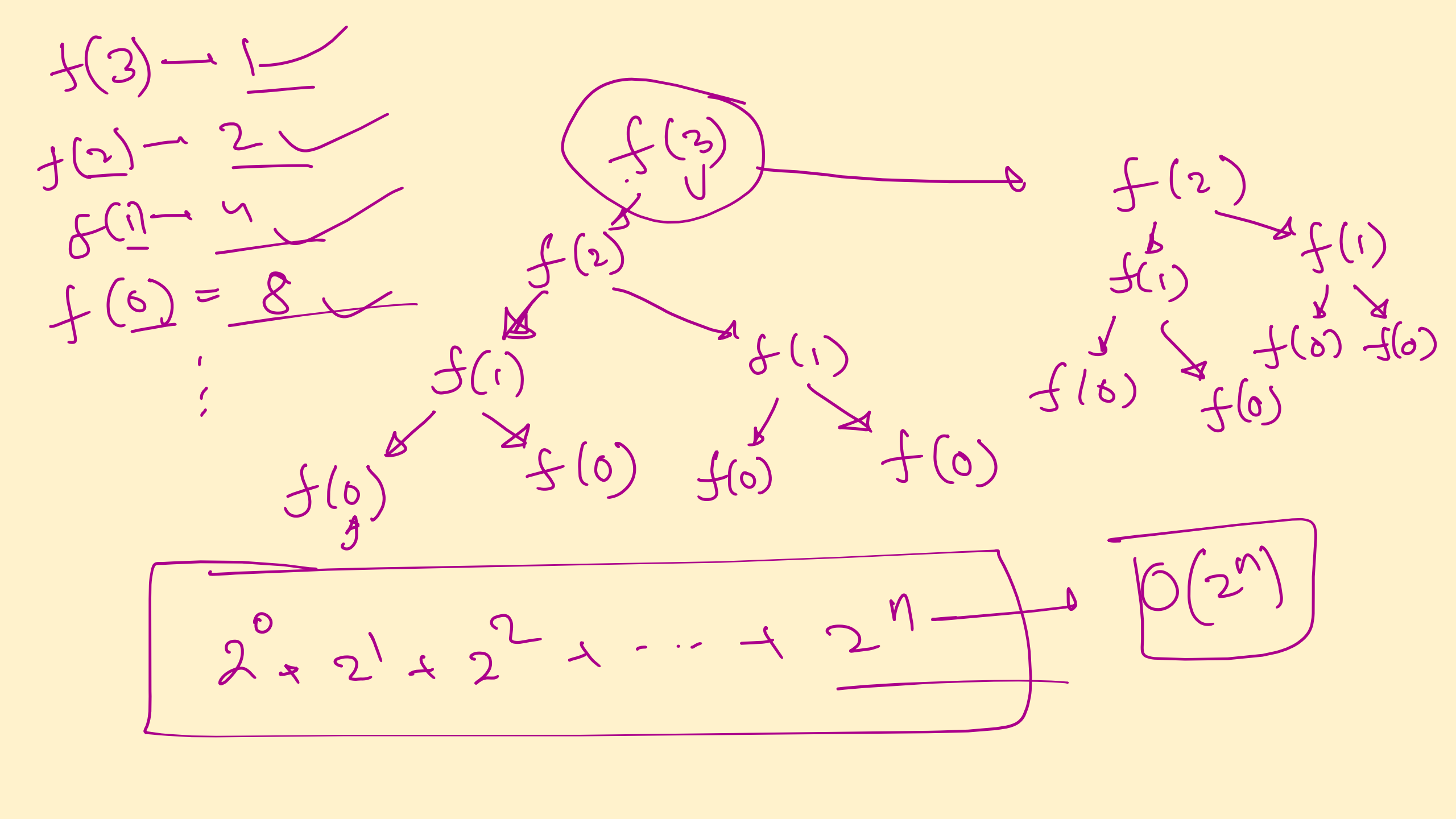
```
void recurse(int n) {
    if (n == 0) return;
    recurse(n-1);
    recurse(n-1);
}
```

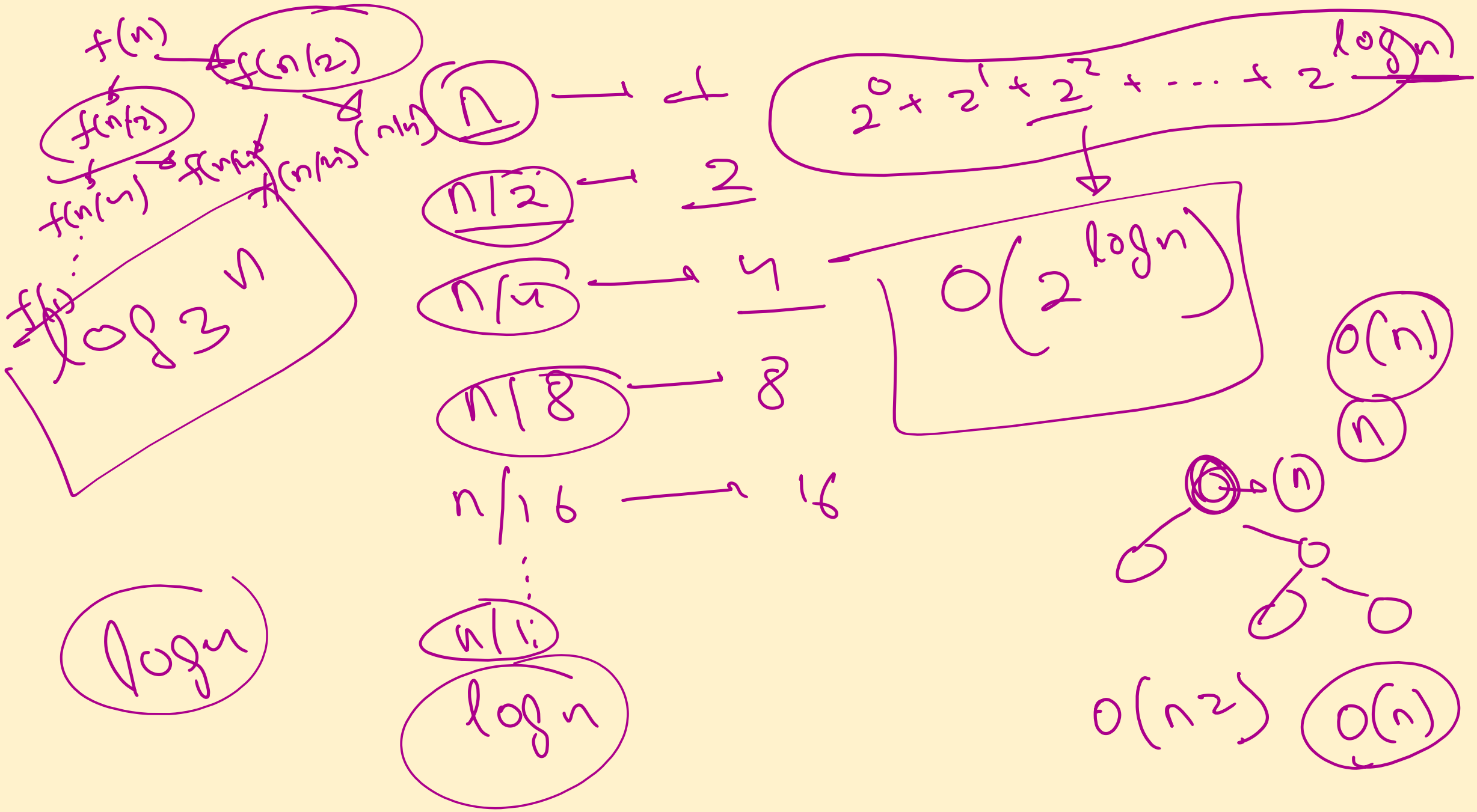
```
void recurse(int n) {
    if (n == 0) return;
    recurse(n/2);
}
```

```
void recurse(int n) {
    if (n == 0) return;
    recurse(n/2);
    recurse(n/2);
}
```


n - 1
n - 2
n - 3
n - 4
f(n)
f(n-1)
f(n-2)
f(n-2)
f(n-3)
f(n-3)
f(n-3)
f(0)
f(0)
f(1)
f(0)
1
2 times
2^n
twice
n nodes


64
n/2^0
32
n/2^1
16
n/2^2
8
4
2
1
x steps
n/2^x
x^th
log2n
f(n)
f(n/2)
f(n/4)
f(n/2^x)
f(1)
n/2^x = 1
n = 2^x
x = log2^n
1/2^x0
f(x=1)
f(0)


f(3) → 1
f(2) → 2
f(1) → 4
f(0) = 8
f(3)
f(2)
f(1)
f(1)
f(0)
f(0)
f(0)
f(0)
f(2)
f(1)
f(1)
f(0)
f(0)
f(1)
f(0)
f(0)
$2^0 + 2^1 + 2^2 + \cdots + 2^n$
$O(2^n)$


f(n)
f(n/2)
f(n/2)
f(n/4)
f(n/4)
f(n/4)
log2 n
logn
n → 1
n/2 → 2
n/4 → 4
n/8 → 8
n/16 → 16
n/1:
logn
2^0 + 2^1 + 2^2 + ... + 2^logn
O(2^logn)
O(n)
n
n
O(n2)
O(n)

Thank You ☺